# 18 | 容器网络配置（3）：容器中的网络乱序包怎么这么高？

2020-12-25 李程远

容器实战高手课 进入课程 >

**讲述：李程远**
时长 14:22 大小 13.16M

你好，我是程远。这一讲，我们来聊一下容器中发包乱序的问题。

这个问题也同样来自于工作实践，我们的用户把他们的应用程序从物理机迁移到容器之后，从网络监控中发现，容器中数据包的重传的数量要比在物理机里高了不少。

在网络的前面几讲里，我们已经知道了容器网络缺省的接口是 veth，veth 接口都是成对使用的。容器通过 veth 接口向外发送数据，首先需要从 veth 的一个接口发送给跟它成对的另一个接口。

那么这种接口会不会引起更多的网络重传呢？如果会引起重传，原因是什么，我们又要如何解决呢？接下来我们就带着这三个问题开始今天的学习。

## 问题重现

我们可以在容器里运行一下 iperf3 命令，向容器外部发送一下数据，从 iperf3 的输出"Retr"列里，我们可以看到有多少重传的数据包。

比如下面的例子里，我们可以看到有 162 个重传的数据包。

复制代码

```
# iperf3 -c 192.168.147.51
Connecting to host 192.168.147.51, port 5201
[  5] local 192.168.225.12 port 51700 connected to 192.168.147.51 port 5201
[ ID] Interval           Transfer     Bitrate                         Retr    C
[  5]   0.00-1.00   sec  1001 MBytes  8.40 Gbits/sec  162    192 KBytes
…
- - - - - - - - - - - - - - - - - - - - - - - - -
[ ID] Interval           Transfer     Bitrate         Retr
[  5]   0.00-10.00  sec  9.85 GBytes  8.46 Gbits/sec  162             sender
[  5]   0.00-10.04  sec  9.85 GBytes  8.42 Gbits/sec                  receiver

iperf Done.
```

**网络中发生了数据包的重传，有可能是数据包在网络中丢了，也有可能是数据包乱序导致的。**那么，我们怎么来判断到底是哪一种情况引起的重传呢？

最直接的方法就是用 tcpdump 去抓包，不过对于大流量的网络，用 tcpdump 抓包瞬间就会有几个 GB 的数据。可是这样做的话，带来的额外系统开销比较大，特别是在生产环境中这个方法也不太好用。

所以这里我们有一个简单的方法，那就是运行 netstat 命令来查看协议栈中的丢包和重传的情况。比如说，在运行上面的 iperf3 命令前后，我们都在容器的 Network Namespace 里运行一下 netstat 看看重传的情况。

我们会发现，一共发生了 162 次（604-442）快速重传（fast retransmits），这个数值和 iperf3 中的 Retr 列里的数值是一样的。

复制代码

```
-bash-4.2# nsenter -t 51598 -n netstat -s | grep retran
    454 segments retransmited
    442 fast retransmits
```

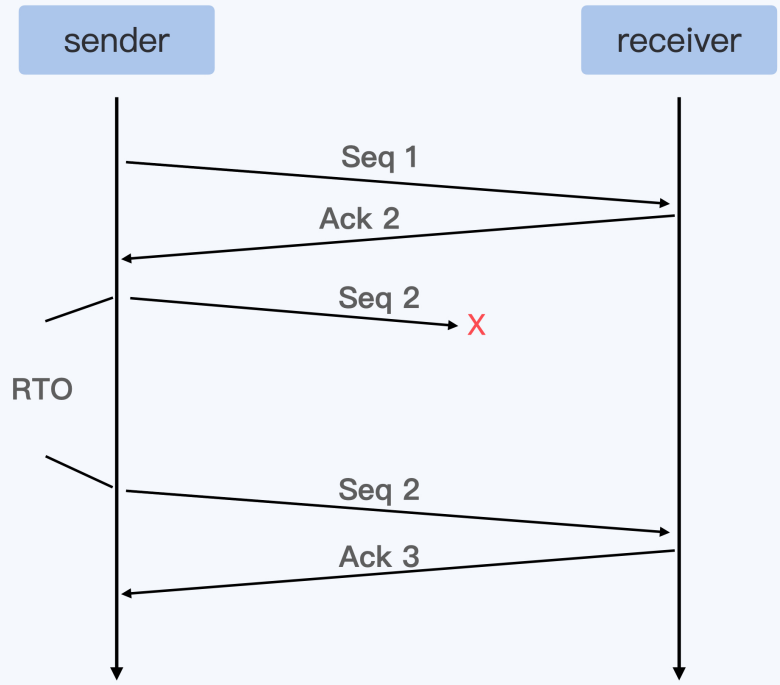
```
-bash-4.2# nsenter -t 51598 -n netstat -s | grep retran
    616 segments retransmited
    604 fast retransmits
```

## 问题分析

### 快速重传（fast retransmit）

在刚才的问题重现里，我们运行 netstat 命令后，统计了快速重传的次数。那什么是快速重传（fast retransmit）呢？这里我给你解释一下。

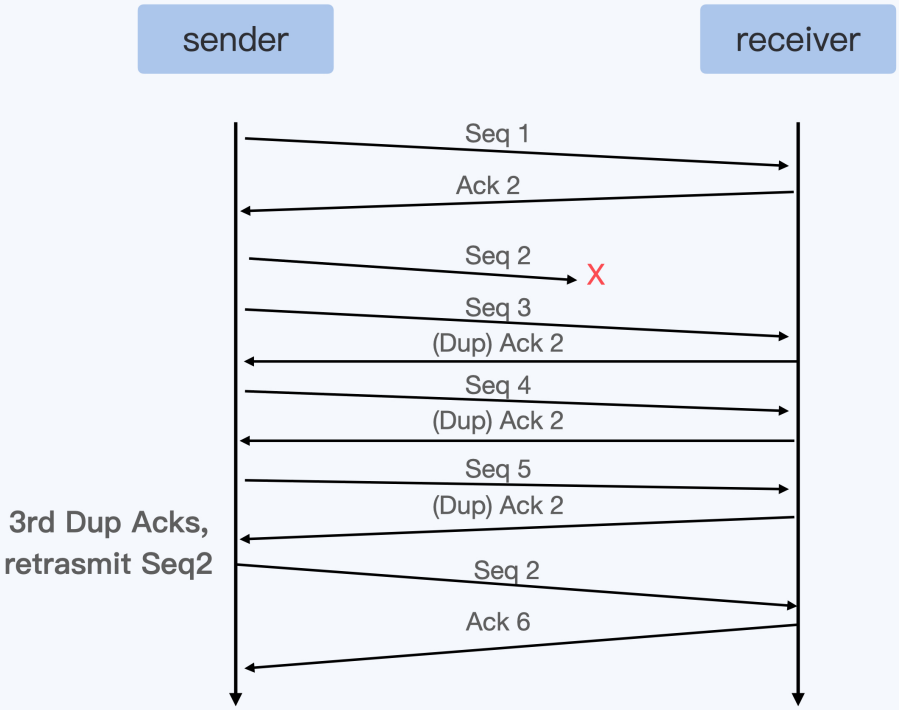
我们都知道 TCP 协议里，发送端（sender）向接受端（receiver）发送一个数据包，接受端（receiver）都回应 ACK。如果超过一个协议栈规定的时间（RTO），发送端没有收到 ACK 包，那么发送端就会重传（Retransmit）数据包，就像下面的示意图一样。




不过呢，这样等待一个超时之后再重传数据，对于实际应用来说太慢了，所以 TCP 协议又定义了快速重传 （fast retransmit）的概念。它的基本定义是这样的：**如果发送端收到 3 个重复的 ACK，那么发送端就可以立刻重新发送 ACK 对应的下一个数据包。**

就像下面示意图里描述的那样，接受端没有收到 Seq 2 这个包，但是收到了 Seq 3–5 的数据包，那么接收端在回应 Ack 的时候，Ack 的数值只能是 2。这是因为按顺序来说收到 Seq 1 的包之后，后面 Seq 2 一直没有到，所以接收端就只能一直发送 Ack 2。

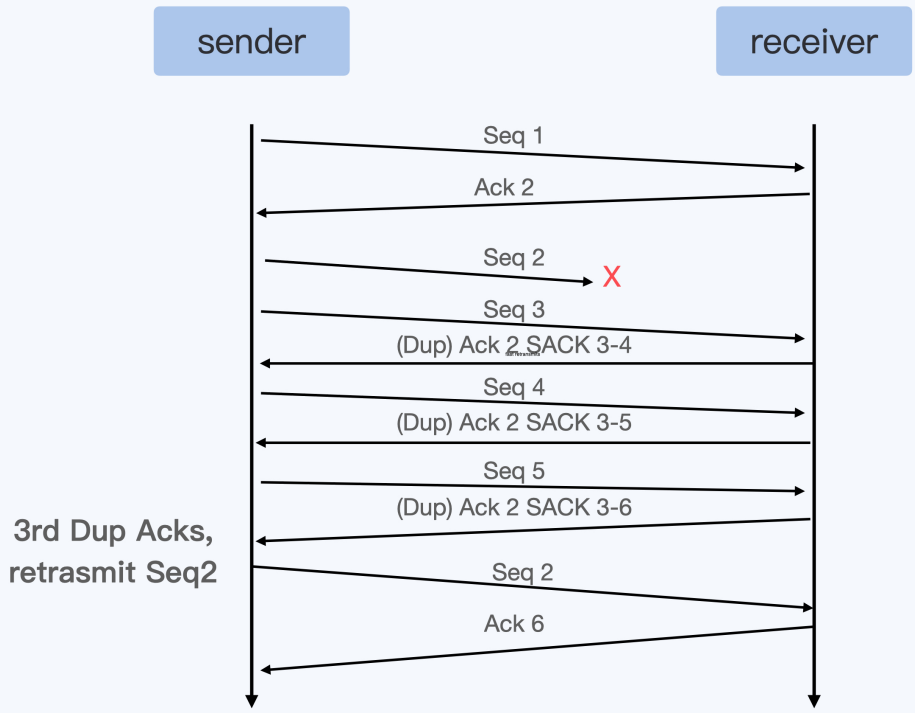
那么当发送端收到 3 个重复的 Ack 2 后，就可以马上重新发送 Seq 2 这个数据包了，而不用再等到重传超时之后了。




虽然 TCP 快速重传的标准定义是需要收到 3 个重复的 Ack，不过你会发现在 Linux 中常常收到一个 Dup Ack（重复的 Ack）后，就马上重传数据了。这是什么原因呢?

这里先需要提到 **SACK** 这个概念，SACK 也就是选择性确认（Selective Acknowledgement）。其实跟普通的 ACK 相比呢，SACK 会把接收端收到的所有包的序列信息，都反馈给发送端。

你看看下面这张图，就能明白这是什么意思了。




那有了 SACK，对于发送端来说，在收到 SACK 之后就已经知道接收端收到了哪些数据，没有收到哪些数据。

在 Linux 内核中会有个判断（你可以看看下面的这个函数），大概意思是这样的：如果在接收端收到的数据和还没有收到的数据之间，两者数据量差得太大的话（超过了 reordering*mss_cache），也可以马上重传数据。

这里你需要注意一下，**这里的数据量差是根据 bytes 来计算的，而不是按照包的数目来计算的，所以你会看到即使只收到一个 SACK，Linux 也可以重发数据包。**

复制代码

```
static bool tcp_force_fast_retransmit(struct sock *sk)
{
        struct tcp_sock *tp = tcp_sk(sk);

        return after(tcp_highest_sack_seq(tp),
                     tp->snd_una + tp->reordering * tp->mss_cache);
}
```

好了，了解了快速重传的概念之后，我们再来看看，如果 netstat 中有大量的"fast retransmits"意味着什么？

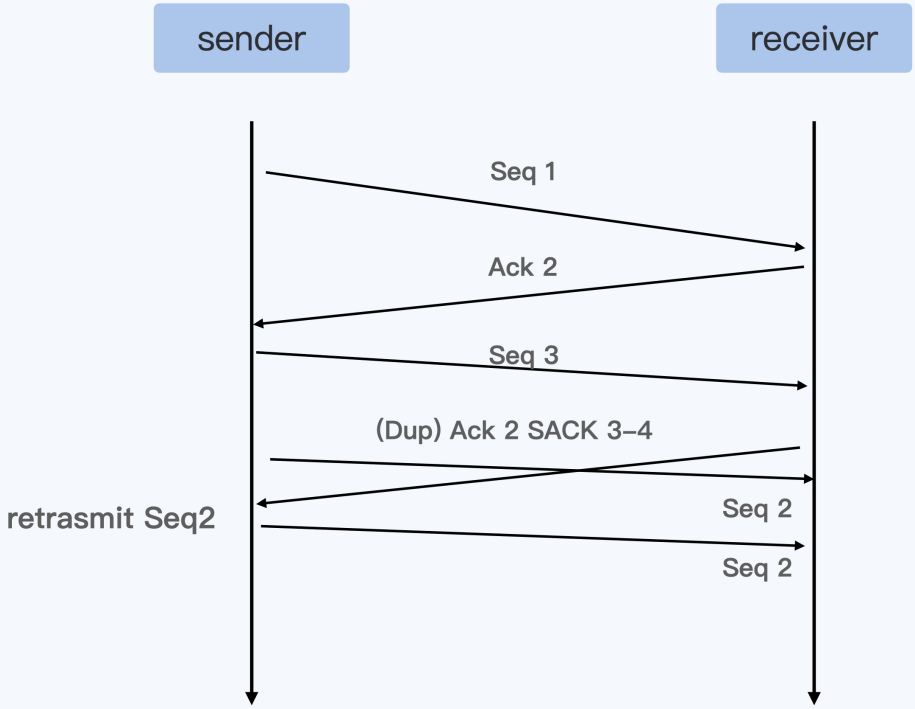
如果你再用 netstat 查看"reordering"，就可以看到大量的 SACK 发现的乱序包。

复制代码

```
-bash-4.2# nsenter -t 51598 -n netstat -s  | grep reordering
    Detected reordering 501067 times using SACK
```

**其实在云平台的这种网络环境里，网络包乱序 +SACK 之后，产生的数据包重传的量要远远高于网络丢包引起的重传。**

比如说像下面这张图里展示的这样，Seq 2 与 Seq 3 这两个包如果乱序的话，那么就会引起 Seq 2 的立刻重传。




## Veth 接口的数据包的发送

现在我们知道了网络包乱序会造成数据包的重传，接着我们再来看看容器的 veth 接口配置有没有可能会引起数据包的乱序。

在上一讲里，我们讲过通过 veth 接口从容器向外发送数据包，会触发 peer veth 设备去接收数据包，这个接收的过程就是一个网络的 softirq 的处理过程。

在触发 softirq 之前，veth 接口会模拟硬件接收数据的过程，通过 enqueue_to_backlog() 函数把数据包放到某个 CPU 对应的数据包队列里（softnet_data）。

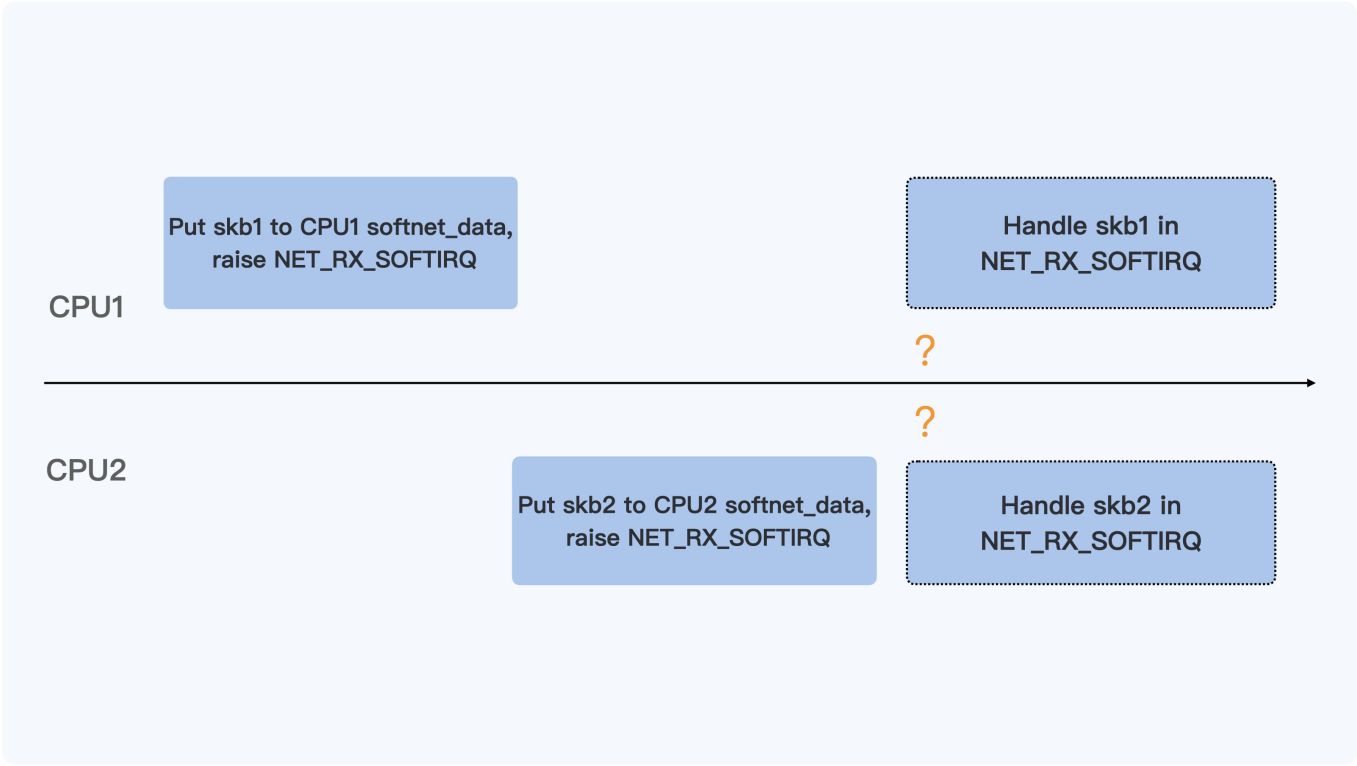
复制代码

```
static int netif_rx_internal(struct sk_buff *skb)
{
        int ret;

        net_timestamp_check(netdev_tstamp_prequeue, skb);

        trace_netif_rx(skb);

#ifdef CONFIG_RPS
        if (static_branch_unlikely(&rps_needed)) {
                struct rps_dev_flow voidflow, *rflow = &voidflow;
                int cpu;

                preempt_disable();
                rcu_read_lock();

                cpu = get_rps_cpu(skb->dev, skb, &rflow);
                if (cpu < 0)
                        cpu = smp_processor_id();

                ret = enqueue_to_backlog(skb, cpu, &rflow->last_qtail);

                rcu_read_unlock();
                preempt_enable();
        } else
#endif
        {
                unsigned int qtail;

                ret = enqueue_to_backlog(skb, get_cpu(), &qtail);
                put_cpu();
        }
        return ret;
}
```

从上面的代码，我们可以看到，在缺省的状况下（也就是没有 RPS 的情况下），enqueue_to_backlog() 把数据包放到了“当前运行的 CPU”（get_cpu()）对应的数据队列中。如果是从容器里通过 veth 对外发送数据包，那么这个“当前运行的 CPU”就是容器中发送数据的进程所在的 CPU。

对于多核的系统，这个发送数据的进程可以在多个 CPU 上切换运行。进程在不同的 CPU 上把数据放入队列并且 raise softirq 之后，因为每个 CPU 上处理 softirq 是个异步操作，所以两个 CPU network softirq handler 处理这个进程的数据包时，处理的先后顺序并不能保证。

所以，veth 对的这种发送数据方式增加了容器向外发送数据出现乱序的几率。




## RSS 和 RPS

那么对于 veth 接口的这种发包方式，有办法减少一下乱序的几率吗？

其实，我们在上面 netif_rx_internal() 那段代码中，有一段在"#ifdef CONFIG_RPS"中的代码。

我们看到这段代码中在调用 enqueue_to_backlog() 的时候，传入的 CPU 并不是当前运行的 CPU，而是通过 get_rps_cpu() 得到的 CPU，那么这会有什么不同呢？这里的 RPS 又是什么意思呢？

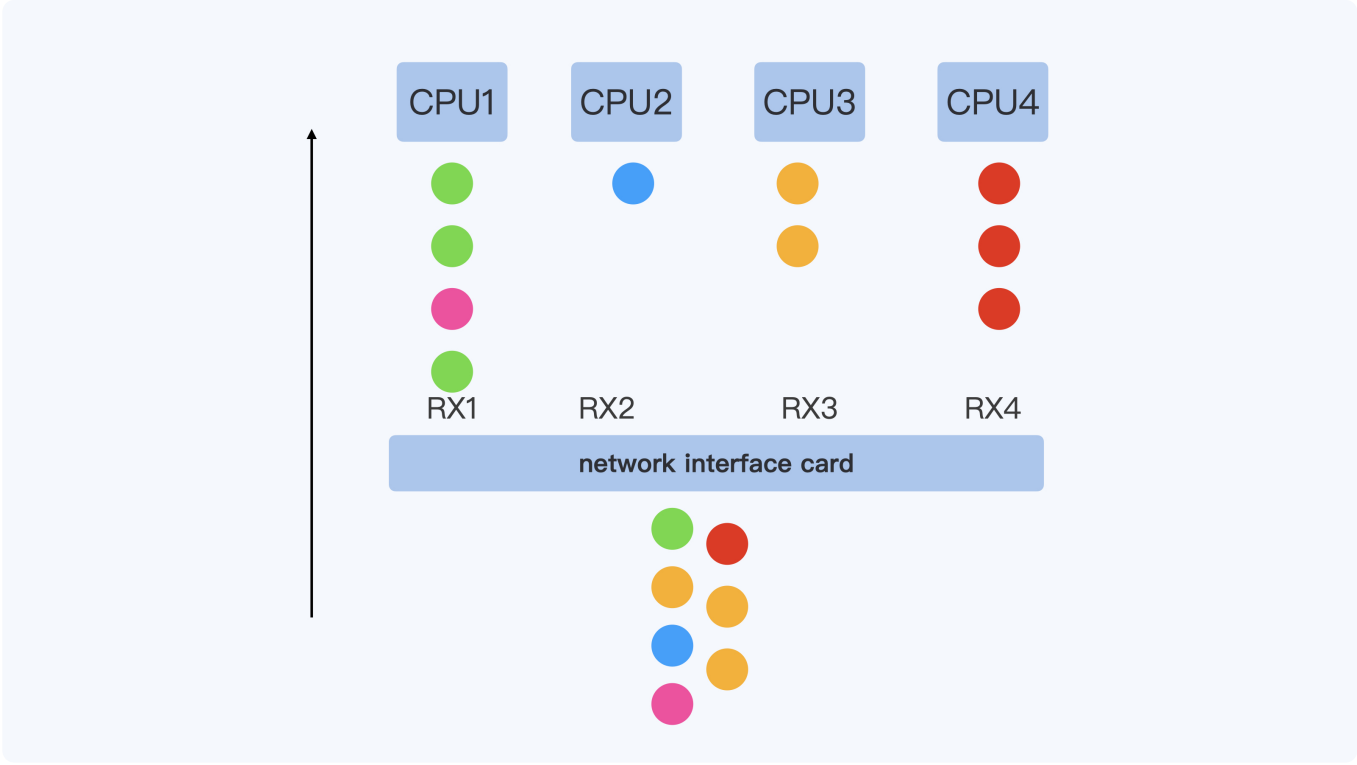
要解释 RPS 呢，需要先看一下 RSS，这个 RSS 不是我们之前说的内存 RSS，而是和网卡硬件相关的一个概念，它是 Receive Side Scaling 的缩写。

现在的网卡性能越来越强劲了，从原来一条 RX 队列扩展到了 N 条 RX 队列，而网卡的硬件中断也从一个硬件中断，变成了每条 RX 队列都会有一个硬件中断。

每个硬件中断可以由一个 CPU 来处理，那么对于多核的系统，多个 CPU 可以并行的接收网络包，这样就大大地提高了系统的网络数据的处理能力.

同时，在网卡硬件中，可以根据数据包的 4 元组或者 5 元组信息来保证同一个数据流，比如一个 TCP 流的数据始终在一个 RX 队列中，这样也能保证同一流不会出现乱序的情况。

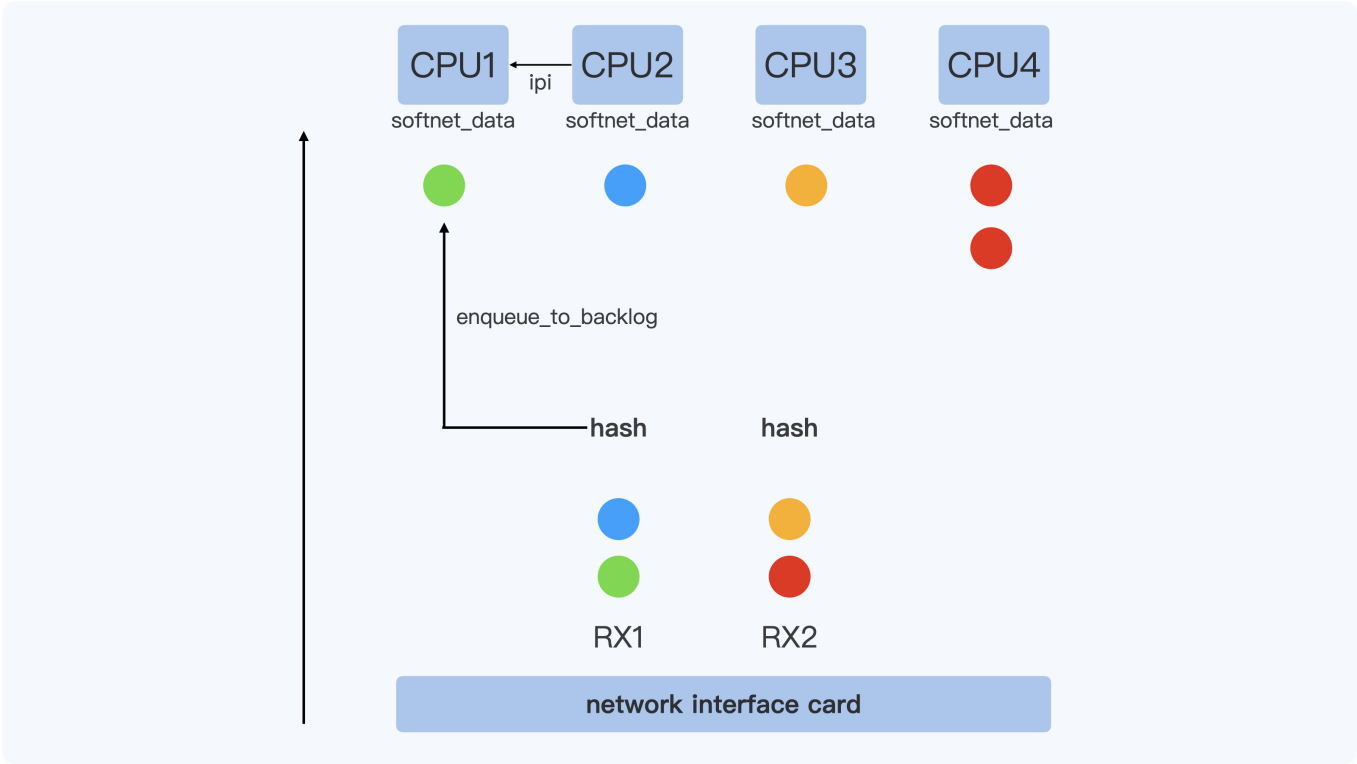
下面这张图，大致描述了一下 RSS 是怎么工作的。




RSS 的实现在网卡硬件和驱动里面，而 RPS（Receive Packet Steering）其实就是在软件层面实现类似的功能。它主要实现的代码框架就在上面的 netif_rx_internal() 代码里，原理也不难。

就像下面的这张示意图里描述的这样：在硬件中断后，CPU2 收到了数据包，再一次对数据包计算一次四元组的 hash 值，得到这个数据包与 CPU1 的映射关系。接着会把这个数据包放到 CPU1 对应的 softnet_data 数据队列中，同时向 CPU1 发送一个 IPI 的中断信号。

这样一来，后面 CPU1 就会继续按照 Netowrk softirq 的方式来处理这个数据包了。




RSS 和 RPS 的目的都是把数据包分散到更多的 CPU 上进行处理，使得系统有更强的网络包处理能力。在把数据包分散到各个 CPU 时，保证了同一个数据流在一个 CPU 上，这样就可以减少包的乱序。

明白了 RPS 的概念之后，我们再回头来看 veth 对外发送数据时候，在 enqueue_to_backlog() 的时候选择 CPU 的问题。显然，如果对应的 veth 接口上打开了 RPS 的配置以后，那么对于同一个数据流，就可以始终选择同一个 CPU 了。

其实我们打开 RPS 的方法挺简单的，只要去 /sys 目录下，在网络接口设备接收队列中修改队列里的 rps_cpus 的值，这样就可以了。rps_cpus 是一个 16 进制的数，每个 bit 代表一个 CPU。

比如说，我们在一个 12CPU 的节点上，想让 host 上的 veth 接口在所有的 12 个 CPU 上，都可以通过 RPS 重新分配数据包。那么就可以执行下面这段命令：

复制代码

```
# cat /sys/devices/virtual/net/veth57703b6/queues/rx-0/rps_cpus
000
# echo fff > /sys/devices/virtual/net/veth57703b6/queues/rx-0/rps_cpus
# cat /sys/devices/virtual/net/veth57703b6/queues/rx-0/rps_cpus
fff
```

## 重点小结

好了，今天的内容讲完了，我们做个总结。我们今天讨论的是容器中网络包乱序引起重传的问题。

由于在容器平台中看到大部分的重传是快速重传（fast retransmits），我们先梳理了什么是快速重传。快速重传的基本定义是：**如果发送端收到 3 个重复的 ACK，那么发送端就可以立刻重新发送 ACK 对应的下一个数据包，而不用等待发送超时。**

不过我们在 Linux 系统上还会看到发送端收到一个重复的 ACK 就快速重传的，这是因为 Linux 下对 SACK 做了一个特别的判断之后，就可以立刻重传数据包。

我们再对容器云平台中的快速重传做分析，就会发现这些重传大部分是由包的乱序触发的。

通过对容器 veth 网络接口进一步研究，我们知道它可能会增加数据包乱序的几率。同时在这个分析过程中，我们也看到了 Linux 网络 RPS 的特性。

**RPS 和 RSS 的作用类似，都是把数据包分散到更多的 CPU 上进行处理，使得系统有更强的网络包处理能力。它们的区别是 RSS 工作在网卡的硬件层，而 RPS 工作在 Linux 内核的软件层。**

在把数据包分散到各个 CPU 时，RPS 保证了同一个数据流是在一个 CPU 上的，这样就可以有效减少包的乱序。那么我们可以把 RPS 的这个特性配置到 veth 网络接口上，来减少数据包乱序的几率。

不过，我这里还要说明的是，RPS 的配置还是会带来额外的系统开销，在某些网络环境中会引起 softirq CPU 使用率的增大。那接口要不要打开 RPS 呢？这个问题你需要根据实际情况来做个权衡。

同时你还要注意，TCP 的乱序包，并不一定都会产生数据包的重传。想要减少网络数据包的重传，我们还可以考虑协议栈中其他参数的设置，比如 /proc/sys/net/ipv4/tcp_reordering。

## 思考题

在这一讲中，我们提到了 Linux 内核中的 tcp_force_fast_retransmit() 函数。那么你可以想想看，这个函数中的 tp->recording 和内核参数 /proc/sys/net/ipv4/tcp_reordering 是什么关系？它们对数据包的重传会带来什么影响？

复制代码

```
static bool tcp_force_fast_retransmit(struct sock *sk)
{
        struct tcp_sock *tp = tcp_sk(sk);

        return after(tcp_highest_sack_seq(tp),
                     tp->snd_una + tp->reordering * tp->mss_cache);
}
```

欢迎你在留言区分享你的思考或疑问。如果学完这一讲让你有所收获，也欢迎转发给你的同事、或者朋友，一起交流探讨。

提建议

© 版权归极客邦科技所有，未经许可不得传播售卖。 页面已增加防盗追踪，如有侵权极客邦将依法追究其法律责任。

上一篇 17 | 容器网络配置（2）：容器网络延时要比宿主机上的高吗？

下一篇 19 | 容器安全（1）：我的容器真的需要privileged权限吗？

## 精选留言 (9)

写留言

**莫名**
2020-12-25

赞，感谢老师，学习了新知识 RPS。

试了下，CONFIG_RPS、rps_cpus 开启时，iperf3 测试仍然存在快速重传情况，没有明显改观。是不是漏了什么？

展开

作者回复: @莫名，
其实除了RSS/RPS外，你还可以去看一下RFS的概念，这个和这一讲的关系不大，并且逻辑要更加复杂，就没有在文档中说明了。

你测试iperf3的环境是什么？如果改了veth的rps_cpus配置，没有效果，那么还要再分析有没有其他情况造成了快速重传。

1 2

**胖达**
2021-01-04

有深度，有什么方式自动配置veth的rps么

展开

作者回复: 现成的自动配置的方法我不知道。我们是在自己实现的cni模块里对veth的rps做配置的。

1

**蒹葭**
2020-12-30

RPS 的配置还是会带来额外的系统开销，在某些网络环境中会引起 softirq CPU 使用率的增大。
老师请教一下这里的某些网络环境指的是什么网络环境？具体增大softirq CPU 使用率的原因是什么呢？

作者回复: RPS会对数据包重新计算hash, 然后把数据包重新分派到新的cpu对应的队列，之后还需要用IPI的中断通知新的cpu, 而在IPI中断之后，就需要再做一次softirq。这样对于高频率收包的情况下， softirq就会明显的增大。
在实际应用的时候，对于物理网络接口，如果已经有了RSS的情况，一般就不需要再打开RPS了。

1

**Geek2014**
2020-12-28

有一点不适很明白： “我们的用户把他们的应用程序从物理机迁移到容器之后，从网络监控中发现，容器中数据包的重传的数量要比在物理机里高了不少”

我理解，这个是因为容器的环境没有设置那个RPS导致的？也就是物理机器如果没设置也会有类似的问题吧

展开

作者回复: 物理机上的网卡和驱动一般都有RSS, 一般不需要RPS。

1

**谢哈哈**

2020-12-25

内核参数tcp_reordering是快速重传的一个初始门限值，到达max tcp_reordering的内核参数大小就会触发快速重传，而tp->reordering是乱序的数据量bytes总大小，触发到达一定大小也会触发快速重传，两者没什么关系，但都是触发快速重传的一个必要条件

展开

1 1

**LWW**

2020-12-25

1.这个网络包的乱序导致的重传最终造成的影响就是网络传输的延迟和吞吐量的下降?
2.这个问题你需要根据实际情况来做个权衡。这个实际情况是指？是说开启rps带来的softirq CPU增高和网络延迟的权衡吗还是说其它方面的权衡？ 这个感觉没有讲清楚，还请老师赐教。

展开

作者回复: @LWW
>1
对的，乱序重传，最终会影响到延时和吞吐量。

>2
RPS会对数据包重新计算hash, 然后把数据包重新分派到新的cpu对应的队列，之后还需要用IPI的中断通知新的cpu, 而在IPI中断之后，就需要再做一次softirq。这样对于高频率收包的情况下， softirq就会明显的增大。
在实际应用的时候，对于物理网络接口，如果已经有了RSS的情况，一般就不需要再打开RPS了。

1 1

**po**

2021-01-14

网络插件(flanneld、calico、ovs)和iptables的关系有点乱，网络插件也有自己的转发表，比如ovs的流表和iptables的规则是什么关系呢?

展开

作者回复: ovs和iptables的规则是独立的。
数据包在协议栈中传递的时候有专门的hook点来处理对应的规则，比如iptables规则在netfilter的pre, in, out等hooks点上处理，ovs规则在ovs相关的device上处理。

**姜姜**
2021-01-08

“在RSS的作用下，每条 RX 队列都会有一个硬件中断，每个硬件中断可以由一个 CPU 来处理。在网卡硬件中，可以根据数据包的 4 元组或者 5 元组信息来保证同一个数据流，比如一个 TCP 流的数据始终在一个 RX 队列中。”

请问老师，这里每一个单独的RX队列的中断是对应一个固定的cpu来处理呢，还是任意...

展开

**上邪忘川**
2020-12-31

rps_cpus 是一个 16 进制的数，每个 bit 代表一个 CPU。那么12个CPU，为啥是FFF？请教一下

作者回复: 一个F是4个bits, 总共三个F， 那么是4 * 3 = 12